MODULE *DistributedLock*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

The set of clients
CONSTANT *Clients*

Client states
CONSTANTS *Active*, *Inactive*

Message types
CONSTANT *LockRequest*, *LockResponse*, *TryLockRequest*, *TryLockResponse*, *UnlockRequest*, *UnlockResponse*

An empty constant
CONSTANT *Nil*

The current lock holder
VARIABLE *lock*

The lock queue
VARIABLE *queue*

The current lock *ID*
VARIABLE *id*

Session states
VARIABLE *sessions*

$serverVars \triangleq \langle lock, id, queue, sessions \rangle$

Client states
VARIABLE *clients*

$clientVars \triangleq \langle clients \rangle$

Client requests
VARIABLE *requests*

Server responses
VARIABLE *responses*

Variable to track the total number of messages sent for use in state constraints when model checking
VARIABLE *messageCount*

$messageVars \triangleq \langle requests, responses, messageCount \rangle$

The invariant checks that:
* No client can hold more than one lock at a time
* No two clients hold a lock with the same *ID*

* The lock is held by an active session

Note that more than one client may believe itself to hold the lock at the same time, *e.g.* if a client's session has expired but the client hasn't been notified, but lock *IDs* must be unique and monotonically increasing.

$$\begin{aligned}
&\mathit{TypeInvariant} \triangleq \\
&\quad \land \forall\, c \in \text{DOMAIN } \mathit{clients} : \mathit{Cardinality}(\mathit{clients}[c].\mathit{locks}) \in 0 \,..\, 1 \\
&\quad \land \forall\, c1,\, c2 \in \text{DOMAIN } \mathit{clients} : c1 \neq c2 \Rightarrow \mathit{Cardinality}(\mathit{clients}[c1].\mathit{locks} \cap \mathit{clients}[c2].\mathit{locks}) = 0 \\
&\quad \land \mathit{lock} \neq \mathit{Nil} \Rightarrow \mathit{sessions}[\mathit{lock}.\mathit{client}].\mathit{state} = \mathit{Active}
\end{aligned}$$

Returns a sequence with the head removed

$$\mathit{Pop}(q) \triangleq \mathit{SubSeq}(q,\, 2,\, \mathit{Len}(q))$$

Sends a request on the given client's channel

$$\begin{aligned}
&\mathit{SendRequest}(m,\, c) \triangleq \\
&\quad \land \mathit{requests}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Append}(\mathit{requests}[c],\, m)] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1
\end{aligned}$$

Sends a response on the given client's channel

$$\begin{aligned}
&\mathit{SendResponse}(m,\, c) \triangleq \\
&\quad \land \mathit{responses}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Append}(\mathit{responses}[c],\, m)] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1
\end{aligned}$$

Removes a request from the given client's channel

$$\begin{aligned}
&\mathit{AcceptRequest}(m,\, c) \triangleq \\
&\quad \land \mathit{requests}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Pop}(\mathit{requests}[c])] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1
\end{aligned}$$

Removes a response from the given client's channel

$$\begin{aligned}
&\mathit{AcceptResponse}(m,\, c) \triangleq \\
&\quad \land \mathit{responses}' = [\mathit{responses} \text{ EXCEPT } ![c] = \mathit{Pop}(\mathit{responses}[c])] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1
\end{aligned}$$

This section models a lock state machine. The state machine supports three types of request:

* *LockRequest* attempts to acquire the lock. If the lock is owned by another process, the request is enqueued until the lock is released.
* *TryLockRequest* attempts to acquire the lock and fails if the lock is already owned by another process.
* *UnlockRequest* attempts to release a lock that's owned by a process.

Additionally, any process's session can be expired, causing any locks held by the session to be released and lock requests enqueued for the session to be removed.

Handles a lock request. If the lock is not currently held by another process, the lock is granted to the client. If the lock is held by a process, the request is added to a queue.

$$\mathit{HandleLockRequest}(m,\, c) \triangleq$$

$\lor \land sessions[c].state \neq Active$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ responses \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock = Nil$
$\quad \land lock' = m$
$\quad \land id' = id + 1$
$\quad \land SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{TRUE},\ id \mapsto id'],\ c)$
$\quad \land \text{UNCHANGED } \langle queue,\ sessions,\ clientVars \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock \neq Nil$
$\quad \land queue' = Append(queue,\ m)$
$\quad \land \text{UNCHANGED } \langle lock,\ id,\ sessions,\ clientVars,\ responses \rangle$

Handles a *tryLock* request. If the lock is not currently held by another process, the lock is granted to the client. Otherwise, the request is rejected.

$HandleTryLockRequest(m,\ c) \triangleq$
$\lor \land sessions[c].state \neq Active$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ responses \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock = Nil$
$\quad \land lock' = m$
$\quad \land id' = id + 1$
$\quad \land SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{TRUE},\ id \mapsto id'],\ c)$
$\quad \land \text{UNCHANGED } \langle queue,\ sessions,\ clientVars \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock \neq Nil$
$\quad \land m.timeout = 1$
$\quad \land queue' = Append(queue,\ m)$
$\quad \land \text{UNCHANGED } \langle clientVars,\ lock,\ id,\ sessions,\ responses \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock \neq Nil$
$\quad \land m.timeout = 0$
$\quad \land SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{FALSE}],\ c)$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars \rangle$

Handles an unlock request. If the lock is currently held by the given client, it will be unlocked. If any client's requests are pending in the queue, the next lock request will be removed from the queue and the lock will be granted to the requesting client.

$HandleUnlockRequest(m,\ c) \triangleq$
$\lor \land sessions[c].state \neq Active$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ responses \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock = Nil$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ responses \rangle$
$\lor \land sessions[c].state = Active$
$\quad \land lock \neq Nil$

$\land\ lock.client = c$
$\land\ lock.id = m.id$
$\land\ \lor\ \land\ Len(queue) > 0$
$\land$ LET $next \triangleq Head(queue)$
IN
$\land\ lock' = next$
$\land\ id' = id + 1$
$\land\ queue' = Pop(queue)$
$\land\ SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{TRUE},\ id \mapsto id'],\ next.client)$
$\land$ UNCHANGED $\langle clientVars,\ sessions\rangle$
$\lor\ \land\ Len(queue) = 0$
$\land\ lock' = Nil$
$\land$ UNCHANGED $\langle clientVars,\ queue,\ id,\ sessions,\ responses\rangle$

Times out a pending *TryLockRequest*. When the request is timed out, the request will be removed from the queue and a response will be sent to the client notifying it of the failure.

$TimeoutTryLock(c) \triangleq$
$\land\ \exists\, i \in \text{DOMAIN}\ queue : queue[i].client = c \land queue[i].type = TryLockRequest$
$\land$ LET $i \triangleq$ CHOOSE $i \in$ DOMAIN $queue : queue[i].client = c \land queue[i].type = TryLockRequestcondition(m$
IN
$\land\ SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{FALSE}],\ c)$
$\land\ queue' = SelectSeq(queue,\ condition)$
$\land$ UNCHANGED $\langle clientVars,\ lock,\ id,\ sessions,\ requests\rangle$

Expires a client's session. If the client currently holds the lock, the lock will be released and the lock will be granted to another client if possible. Additionally, pending lock requests from the client will be removed from the queue.

$ExpireSession(c) \triangleq$
$\land\ sessions[c].state = Active$
$\land\ sessions' = [sessions$ EXCEPT $![c].state = Inactive]$
$\land$ LET $isActive(m) \triangleq sessions'[m.client].state = Active$
IN
IF $lock \neq Nil \land lock.client = c$ THEN
LET $q \triangleq SelectSeq(queue,\ isActive)$
IN
$\lor\ \land\ Len(q) > 0$
$\land\ lock' = Head(q)$
$\land\ id' = id + 1$
$\land\ queue' = Pop(q)$
$\land\ SendResponse([type \mapsto LockResponse,\ acquired \mapsto \text{TRUE},\ id \mapsto id'],\ lock'.client)$
$\land$ UNCHANGED $\langle requests\rangle$
$\lor\ \land\ Len(queue) = 0$
$\land\ lock' = Nil$
$\land\ queue' = \langle\rangle$
$\land$ UNCHANGED $\langle id,\ messageVars\rangle$
ELSE

$\land\ queue' = SelectSeq(queue,\ isActive)$
$\land$ UNCHANGED $\langle lock,\ id,\ messageVars\rangle$
$\land$ UNCHANGED $\langle clientVars\rangle$

This section models a lock client. A client can interact with a lock state machine using three types of requests:
* *Lock* attempts to acquire a lock and blocks until successful or the session expires
* *TryLock* attempts to acquire a lock, failing if the lock is owned by another process
* *Unlock* attempts to release a lock owned by the process

Additionally, a client can assume its session has expired either before or after it actually has. This models the possibility that a client believes its session has expired when it hasn't or that the state machine can expire a client's session without the client knowing.

Sends a lock request to the cluster with a unique *ID* for the client.

$Lock(c) \triangleq$
$\land\ clients[c].state = Active$
$\land\ SendRequest([type \mapsto LockRequest,\ client \mapsto c,\ id \mapsto clients[c].next],\ c)$
$\land\ clients' = [clients$ EXCEPT $![c].next = clients[c].next + 1]$
$\land$ UNCHANGED $\langle serverVars,\ responses\rangle$

Sends a try lock request to the cluster with a unique *ID* for the client.

$TryLock(c) \triangleq$
$\land\ clients[c].state = Active$
$\land\ SendRequest([type \mapsto TryLockRequest,\ client \mapsto c,\ id \mapsto clients[c].next,\ timeout \mapsto 0],\ c)$
$\land\ clients' = [clients$ EXCEPT $![c].next = clients[c].next + 1]$
$\land$ UNCHANGED $\langle serverVars,\ responses\rangle$

Sends a try lock request to the cluster with a timeout and a unique *ID* for the client.

$TryLockWithTimeout(c) \triangleq$
$\land\ clients[c].state = Active$
$\land\ SendRequest([type \mapsto TryLockRequest,\ client \mapsto c,\ id \mapsto clients[c].next,\ timeout \mapsto 1],\ c)$
$\land\ clients' = [clients$ EXCEPT $![c].next = clients[c].next + 1]$
$\land$ UNCHANGED $\langle serverVars,\ responses\rangle$

Sends an unlock request to the cluster if the client is active and current holds a lock.

$Unlock(c) \triangleq$
$\land\ clients[c].state = Active$
$\land\ Cardinality(clients[c].locks) > 0$
$\land\ SendRequest([type \mapsto UnlockRequest,\ client \mapsto c,\ id \mapsto$ CHOOSE $l \in clients[c].locks : \text{TRUE}],\ c)$
$\land\ clients' = [clients$ EXCEPT $![c].locks = clients[c].locks \setminus \{$CHOOSE $l \in clients[c].locks : \text{TRUE}\}]$
$\land$ UNCHANGED $\langle serverVars,\ responses\rangle$

Handles a lock response from the cluster. If the client's session is expired, the response is ignored. If the lock was acquired successfully, it's added to the client's lock set.

$HandleLockResponse(m,\ c) \triangleq$

$\land \lor \land clients[c].state = Inactive$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ requests \rangle$
$\quad \lor \land clients[c].state = Active$
$\quad \land m.acquired$
$\quad \land clients' = [clients \text{ EXCEPT } ![c].locks = clients[c].locks \cup \{m.id\}]$
$\quad \land \text{UNCHANGED } \langle serverVars,\ requests \rangle$
$\quad \lor \land clients[c].state = Active$
$\quad \land \neg m.acquired$
$\quad \land \text{UNCHANGED } \langle clientVars,\ serverVars,\ requests \rangle$

Closes a client's expired session. This is performed in a separate step to model the time between the cluster expiring a session and the client being notified. A client can close its session either before or after it's expired by the cluster. Once the client believes its session has expired, its locks are removed, meaning a client can also believe itself to hold a lock after its session has expired in the cluster.

$CloseSession(c) \triangleq$
$\land clients[c].state = Active$
$\land clients' = [clients \text{ EXCEPT } ![c].state = Inactive,$
$\qquad\qquad\qquad\qquad\qquad\qquad ![c].locks = \{\}]$
$\land \text{UNCHANGED } \langle serverVars,\ messageVars \rangle$

Receives a request 'm' from the client 'c' to the cluster.

$ReceiveRequest(m,\ c) \triangleq$
$\land \lor \land m.type = LockRequest$
$\quad \land HandleLockRequest(m,\ c)$
$\quad \lor \land m.type = TryLockRequest$
$\quad \land HandleTryLockRequest(m,\ c)$
$\quad \lor \land m.type = UnlockRequest$
$\quad \land HandleUnlockRequest(m,\ c)$
$\land AcceptRequest(m,\ c)$

Receives a response 'm' from the cluster to the client 'c'.

$ReceiveResponse(m,\ c) \triangleq$
$\land \lor \land m.type = LockResponse$
$\quad \land HandleLockResponse(m,\ c)$
$\land AcceptResponse(m,\ c)$

Initial state predicate

$Init \triangleq$
$\land requests = [c \in Clients \mapsto \langle\rangle]$
$\land responses = [c \in Clients \mapsto \langle\rangle]$
$\land messageCount = 0$
$\land lock = Nil$

$\land\ queue = \langle\rangle$
$\land\ id = 0$
$\land\ clients = [c \in Clients \mapsto [state \mapsto Active,\ locks \mapsto \{\},\ next \mapsto 1]]$
$\land\ sessions = [c \in Clients \mapsto [state \mapsto Active]]$

Next state predicate

$Next \triangleq$
$\lor \exists\, c \in \text{DOMAIN } clients : \exists\, i \in \text{DOMAIN } requests[c] : ReceiveRequest(requests[c][i],\ c)$
$\lor \exists\, c \in \text{DOMAIN } clients : \exists\, i \in \text{DOMAIN } responses[c] : ReceiveResponse(responses[c][i],\ c)$
$\lor \exists\, c \in \text{DOMAIN } clients : Lock(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : TryLock(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : TryLockWithTimeout(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : Unlock(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : TimeoutTryLock(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : ExpireSession(c)$
$\lor \exists\, c \in \text{DOMAIN } clients : CloseSession(c)$

The specification includes the initial state predicate and the next state

$Spec \triangleq Init \land \Box[Next]_{\langle serverVars,\ clientVars,\ messageVars\rangle}$

* Modification History
* Last modified Sun *Jan* 28 19:11:36 *PST* 2018 by *jordanhalterman*
* Created *Fri Jan* 26 13:12:01 *PST* 2018 by *jordanhalterman*